Acknowledgement:

* <https://hackernoon.com/14-patterns-to-ace-any-coding-interview-question-c5bb3357f6ed>

1: Sliding Window

* The Sliding Window Pattern is used to perform a required operation on a specific window size of a given array or linked list, such as finding the longest subarray contraining all 1s. Sliding windows start from the 1st element and keep shifting right by one element and adjust the length of the window accoring to the problem that you are solving. In some cases, the window size remains constant and in other cases, the sizes grows or shrinks

Diagram

Description automatically generated

Following are some ways you can identify that the given problem might require a sliding window:

* The problem input is a linear data structure such as a linked list, array, or string
* You’re asked to find the longest/shorted substring, subarray, or a desired value

Common problems you use the sliding window pattern with:

* Maxinum sum subarray of size ‘K’ (easy)
* Longest substring with ‘K’ distinct characters (medium)
* String anagrams (hard)

340

Solution Article

Approach 1: Sliding Window + HashMap

Intuition

We could take some inspiration from a simpler problem called longest subtring with at most 2 distinct characters.

To solve the problem in one pass let’s use here sliding window approach with 2 set pointers left and right serving as the windown boundaries.

The idea is to set both pointers in the position 0 and then move right pointer to the right while the window contains not more than k distinct characters.

The idea is to set both pointers in the position 0 and then move right pointer to the right while the windown contains not more than k distinct characters. If at some point, we’ve got k + 1 distinct characters, let move left pointer to keep not more than k+1 distinct characters in the window.

Table

Description automatically generated

Table

Description automatically generated

* To move sliding window along the string
* to keep not more than k distinct characters in the window
* and to update max subtring length at each step.

Next question:

* how to move the left pointer to keep only k distinct characters in the string?

Let’s use for this purpose hashmap containing all characters in the sliding window as keys and their rightmost positions as values. At each moment, this hashmap could contain not more than k + 1 elements

A picture containing table

Description automatically generated

A picture containing table

Description automatically generated

For example, using this hashmap one knows that the rightmost position of character O in “LOVELEE” window is 1 and so one has to move left pointer in the position 1+1 = 2 to exclude the character O from the sliding window.

Algorithm:

Now one could write down the algortihm

* return 0 if the string is empty or k is equal to zero
* Set both pointers in the beginning of the string left = 0 and right = 0 and init max substring length max\_len = 1
* While right pointer is less than N:
  + Add the current character s[right] in the hashmap and move right pointer to the right.
  + If hashmap contains k + 1 distinct characters, remove the left most character from the hashmap and move the left pointer so that sliding window contains again k distinct characters only
  + Update max\_len

Complexity Analysis

Do we have here the best possible time complextiy O(N) as it was for more simple problem with at most 2 distint characters?

For the best case when input string contains not more than k distinct characters the answer is yes. It’s only one pass along the string with N characters and the time complexity is O(N).

For the worst case when the input string contains n distinct characters, the answer is no.

Abcdef => N distinct characters.

In that case, at each step one uses O(k) time to find a minimum value in the hashmap with k elements and so the overall time complexity is O(Nk).

* Time complexity: O(N) in the best case of k distinct characters in the string and O(Nk) in the worst case of N distinct characters in the string.
* Space complexity: O(k) since additional space is used only for a hashmap with at most k + 1 elements.

Not more than k distinct characters:

Aaabbcccc k = 2

Takeaways:

- need a hashMap of key/value: character: last seen position

- slide window.

- move the left pointer after deleting a character: left = leftMost + 1

* Look at the solution for improvement

Graphical user interface, text, application

Description automatically generated

Don’t store the index, instead, store the count of each character, once right point (count == 0), we increase the number of distinct character by 1, left pointer keeps once count reaches 0, we decrease it.

Kadane’s Algorithm

Kadane’s original algorithm solves the problem version when empty subarrays are admitted. It scans the given array A[1…n] from left to right. In the jth step, it computes the subarray with the largest sum ending at j; this sum is maintained in variable current\_sum. Moreover, it computes the subarray with the largest sum anywhere in A[1…j], maintained in variable best\_sum, and easily obtained as the maxinum of all values of current\_sum seen so far, cf. line 7 of the algorithm.

As a loop invariant, in the jth step, the old value of current\_sum holds the maximum over all i in {1, …., j} of the sum A[i] + … + A[j-1]. Therefore, current\_sum + A[j] is the maximum over all i in {1, ….j} of the sum A[i] + … + A[j]. To extend the latter maxinm to cover also the case i = j + 1, it is sufficient to consider also the empty subarray A[j + 1 … j]. This is done in line 6 by assigning max (0, current\_sum + A[j]) as the new value of current\_sum, which after that holds the maxinum over all i in {1, …, j+1} of the sum A[i] + … + A[j]

For the variant of the problem which disallows empty subarrays, best\_sum should be initialized to negative infinity instead and also in the for loop current\_sum should be updates as max(x, current\_sum + x). In that case, if the input contains no positive element, the return value is that of the largest element (i.e, the value closet to 0), or negative infinity if the input was empty.

maxSubArray with Kadane’s Algorithm

* Initialize 2 integer variables. Set both of them equal to the first value in the array.
  + currentSubarray will keep the running count of the current subarray we are focusing on
  + maxSubarray will be our final return value. Continously update it whenever we find a bigger subarray
* Iterate through the array, starting with the 2nd element (as we used the first element to initialize our variable).
  + For each number, add it to the currentSubarray we are building. Remember to update maxSubarray everytime we find a new maxinum
* Return maxSubarray.